All I Ever Needed to Know About Programming, I Learned From Re-writing Classic Arcade Games

Katrin Becker
Graduate Division of Educational Research,
Faculty of Education
University of Calgary
beckerk@ucalgary.ca

J. R. Parker
Digital Media Laboratory
University of Calgary
jparker@ucalgary.ca

Abstract

The quest for interesting, engaging, yet doable programming assignments is an ongoing one. Authentic, realistic examples have often been drawn from business, and games have often been overlooked as being too narrow in scope. This paper explains why computer games, especially classic arcade games are ideal vehicles for learning to program. Games are important popular cultural objects that should not be dismissed. Indeed, classic arcade games embody virtually all of the components necessary for a thorough grounding in software design, and can easily be applied to many advanced topics. Various classic arcade games are examined to demonstrate where they connect with computer science pedagogy.

1. Introduction

According to the Computing Research Association's report of May 2005, enrollment in computing science programs has declined an average of 39% since 2000 [1]. This trend shows no signs of abating, especially with the current trend towards outsourcing of high tech jobs, primarily to the Far East. Although there is an unarguable drop in demand for high tech professionals, it is predicted that the drop in demand will be considerably less than the drop in enrollments [2]. This trend has many consequences, among them: How will we fill the need for programmers of all kinds and games programmers in particular in the near future? How can we interest freshman in choosing computer science? This is still where most of our games programmers begin. Part of the answer for both of these questions is to teach programming by having students write games, and classic arcade games are especially suited for this role. Others are also recognizing the value of games in the curriculum: Microsoft Research chose games curricula as one of their two major focuses for funding in 2004 [3].

2. Traditional Project Fare

As the twenty-first century begins to unfold, we continue the frequently intense reflection on the previous century, and one of the areas under scrutiny is formal education. The Taylorian model of education, that is, the scientific management of learning has, among other things, resulted in widespread mathematical sequencing of curriculum into uniform, incremental steps. This has also resulted in the notion that the 'proper' way to teach science is in a step-wise fashion, beginning with very simple problems and examples, and progressing slowly to more complex ones [4], often culminating in "the capstone course." Perhaps it is no coincidence that the term used to describe such a course comes from masonry – a medium not renowned for its malleability.

In Computer Science, this serial, graduated order of instruction resulted in novice 'projects' that involved writing programs that did little more than sort lists of simple data (like names). In more recent times this has been updated to include writing programs to manage employee records – maybe even implementing graphical interfaces for the user menus. But, guess what? They are still sorting lists, only now the data is more complex. The actual problem is much the same.

Little by little, educators are beginning to question the absoluteness of this serial, graduated order and considering more dynamic ones. Until fairly recently, this has been "a hidden but dominant aspect of contemporary curriculum,

from first grade through college. Only Kindergarten, and doctoral seminars seem able to develop more interactive, dynamic, and complex forms of order." p. 36 [5] These days, terms like *deep learning*, *engagement*, *authentic learning*, and *student-centered* are in vogue. We've begun to accept that we should come up with interesting, engaging, and challenging problems for our students to connect with.

However, we are often at a loss for ideas – we need problems that meet stated objectives, can be solved to a reasonable minimum standard, and yet leave room to challenge the better students. Often additional functionality added to a typical assignment simply involves more code without also requiring greater complexity. This is clearly not effective. We want the problems to be interesting for our students while still forcing them into contact with the necessary content. If building on a student's expertise results in a more engaging problem, we should look seriously at how students spend their time. Problems drawn from accounting or management are common, but seriously, how many students do you know that do accounting as a hobby or pastime?

3. Games and Students

Casual polls of students enrolled in the introductory computer science class at the institution where the authors teach indicate that two thirds to three quarters of all freshmen in computer science became interested in computer science because they play computer and video games. This includes students who have not declared a major, and those who have declared majors in other disciplines. At least half of the students enrolled in the second introductory course express a desire to be involved in the games industry at a level beyond simply being consumers. By and large, students understand games far better than they understand employee records keeping, and 'widget' manufacturing, both of which are sources of favorite entry-level programming problems. Current wisdom implies that learning is most effective when we build on what the learner already knows, and using situations they are familiar with [6, 7]. If they also care about these problems, that is even better. Students care about games. This is not in dispute. The challenge is to demonstrate that games embody many, if not all of the fundamental concepts important to a thorough grounding in computer science.

4. Games and Pedagogy

Gameplay is directly tied to programming: more complex gameplay = more complex and different algorithms to implement. Always. This does not include simply adding more of something: the complexity of the program is roughly the same if you implement 5 lives for Frogger as it would be if you were to implement 5000. There is, however a difference between one and many, and an even greater difference between a frog that can be made to move across the road and be killed by passing cars, and one that gets killed by trucks, but thrown to another location by a car.

Games are highly visual. For one thing, if a student miscounts the number of objects to be drawn on the screen, it is immediately obvious. Students can see their algorithms in action. In other words, on-screen behaviour of the game elements as well as the player control can often be mapped directly onto specific algorithms, and they can be traced *while the algorithm is running*. This kind of immediate feedback is game-like itself, and tends to encourage experimentation with the programs and algorithms. This cannot be said, for example, of doing the monthly payroll run for our employee program.

The importance of program testing is also easy to convey within the context of games. Anyone who has ever played a game, which coincidentally includes virtually all college students, recognize the importance of a software product (i.e. their game) working properly. In fact approaches to testing can easily be related to what students already do while they are playing games ("Try it and see what happens."). Further, the whole notion of 'cheats' and cheat codes in games stems from testing elements that have remained in the game once it ships. This concept is extremely useful when discussing software testing and debugging.

Two common objections to the use of games, aside from those objections that stem from a general disapproval of games as frivolous, are 1) that the graphics (and audio) are irrelevant

anywhere but in a graphics course, and 2) that the event-driven nature of most games situate them in a restricted sub-class of programming problem: i.e. event-driven programming. The conclusion of these objections is that games are applicable to only a select set of courses, and to only a select set of modules within those courses.

To address the first objection, GUI's are great fun, but admittedly complicate an already complex introduction to programming [8], and in the opinion of the authors, should not be the focus of fundamental courses. The ACM curriculum lists GUI programming as a minor element in the introductory sequence [9], and thus de-emphasizing this aspect here will not put students at a disadvantage. Many games can be implemented quite effectively without the use of sophisticated GUIs or graphics and sound, so this is not an essential element of games for our purposes. Further, because of the state of hardware at the time these games were developed, classic arcade games lend themselves especially well to this approach.

Secondly, the event-driven programming can be made optional in virtually all of the games described in this paper by converting them into turn-based play. Event-driven programming is admittedly a difficult concept. Early courses in CS and programming used to avoid event driven code entirely, but more and more students are using Java in first and second year. This language encourages objects early in the presentation sequence, and can involve the use of events early too, as they are connected to SWING and AWT interfaces. Event-driven programming has become an element of the introductory sequence, albeit a small one [9], and games can be created that both include, or exclude event-driven aspects.

5. What's So Special About Arcade Games?

For the purposes of this paper, classic arcade games are defined to include games that were traditionally found in the arcades of the late 70's and early 80's (like *Asteroids!* and *Space Invaders*) as well as games found on early home consoles (like *Pong*). There are three highly significant advantages of these classic arcade games over more modern or custom designed games for the purposes of teaching programming, and they are interconnected.

First, classic arcade games are immediately familiar to most students, and many students already know how to play them. The value of fully understanding how a program is supposed to work is essential to the generation of correct solutions and should not be underestimated. *Minesweeper* for example, while not a classic game, is still a game that almost all students have tried - ***it is part of their PC*** (and has been for a very long time)**.** This puts the problem they are to solve in a context with which they are already familiar [10]. While the implementation of *Minesweeper* is no more difficult than the well-known *Game of Life* (it is in fact marginally simpler), the difference is that they knew *Minesweeper* when they were mere computer users. Writing it themselves and seeing their efforts behave just like the real thing forces them to cross a significant perceptual boundary: they become the "creators". ***They go from experiencing the "magic" to being the "magicians".***

Second, these games were designed when hardware was limited and graphics were, relatively speaking, crude. This means that the internal complexity of the program is, relatively speaking, low. It also means that we can afford to gloss over some of the graphical aspects of games without loss of credibility, or student interest. In fact, many of these games can be designed and built as ASCII games with nothing more than a monochrome text display. Past experience with this approach in a several first year classes indicates that this approach has little negative impact on the students' interest [11]. Implementing a game like *Minesweeper*, for example, is a problem of a complexity that easily compares with any other typical late-term first-year assignment. Conway's *Game of Life* [12] is a long-time favorite, and although students generally enjoy implementing this game, they do not get as excited about it as they do implementing *Minesweeper*.

The third advantage of using arcade games as opposed to using newer commercial games or inventing our own is that multiple excellent working examples exist out there for students to try and play with (for free). This is sometimes cited as an impediment to the use of these games as assignments – namely – students can simply get

working solutions and attempt to pass them off as their own. True. They can. However, this claim is also true for the majority of programming problems that are assigned to undergraduates, regardless of domain. Teaching faculty have been inventing programming assignments for thirty or so years. Chances are high that somebody, somewhere has created a solution to the problem you have posed, and offered it on the web. One effective way to address this problem is to allow students to demonstrate their understanding by having them talk about their own programs and explain various aspects of them as part of the completion requirements. Anyone who didn't write his or her own code would be unable to explain how it works. On the other hand, if they *can* explain, then they have demonstrated that they understand the concepts even if they didn't write all the code themselves, and the goals of the assignment will have been met anyways. We still win.

To summarize, classic arcade games are obviously a part of the game world that was responsible for attracting these students in the first place, which provides an important real-world connection to drive their studies. The games currently have the added bonus of enjoying a renaissance of popularity as 'retro' games. These games were a part of popular culture when game technology was far simpler; so they are part of the culture, yet exist at a level that students can master as programmers. And finally, also because they are part of this domain, there exist plenty of working examples that students can turn to in order to help themselves fully understand the problem, as well as to compare against their own answers.

6. Which Games Teach What Concepts?

The technology embodied in typical digital games means that almost any concept in computer science is represented in some form in some game [13]. Object-oriented programming is the common paradigm used in introductory courses. Virtually all of these game programs can be used to demonstrate object-oriented programming, and make the concepts of polymorphism and inheritance clear and straightforward: of *course* trees and boulders are kinds of obstacles, while potions and bananas are treasures. Obstacles share certain properties and behaviours as a group as well as having individual differences. Treasures also share properties and attributes, yet it is pretty obvious that there is a need for specific differences. This kind of clarity is much harder to achieve and appears much more contrived when using employee records.

Almost all games contain the same basic concepts like list manipulation, subprograms, random number use, error detection and correction, and user interfaces, but some 'classes' of game are more valuable than others for demonstrating specific concepts:

1. Action shooters like *Asteroids!*, *Missile Command* and *Defender* require collisions detection algorithms and distance calculations.
2. *Pac-man* is an excellent maze puzzle requiring path finding and chasing (tracking) algorithms.
3. Puzzle games like *Tetris* and *Qbert* involve 2D geometry, packing algorithms (even though the user/player does the packing, the program must still be able to check the moves) and detection of reasonably complex win-states. These are excellent for practice with the development of efficient condition checking.
4. *Blocks*, *Breakout!*, and *Pong* all require physics (bouncing). These are perhaps the only group of games best designed as real-time, event driven programs. The games themselves are otherwise fairly uncomplicated and so provide an effective balance from a programming perspective.
5. The side-scrolling platform action adventures like *Mario Bros.*, *Donkey Kong*, *Pitfall*, and *Joust* include everything from rudimentary physics to potentially complex inventory and asset management, and various AI techniques.
6. Racing and driving games like *Indy 500* and *Street Racer* feature algorithms in physics, AI and collision detection, but of course can also include all levels (from novice to advanced) of graphics, user interfaces, audio, and 3D animation. [14]
7. Finally, *Zork*, one of the earliest commercially available text-based adventure games, contains all of the fundamental elements of modern role-playing games, but without the multi-million dollar development budget, timelines, and development teams. Role-playing games are

especially useful for practice in parsing, and various AI algorithms.

7. Adding or Removing Complexity

Novice problems are defined in this paper as those that have limited data types and complexity and contain a smaller number of distinct algorithms. They expressly do not require the use of object-oriented constructs like inheritance and polymorphism. They are ideal for learners just beginning to program, they could be implemented in languages like 'C', or even Pascal, and posed in a first course on programming. More complex problems can be simplified by altering the gameplay, or providing 'plug-ins' (routines or utilities that students can use without seeing the code inside) and so can also be turned into novice assignments. For example, the flood-fill algorithm in *Minesweeper* makes the game too complex for many novices because it requires an understanding of recursion, so providing a utility that does it for them reduces the level of complexity to that roughly equivalent to the *Game of Life*. Similarly, maze games and platform action side-scrollers can have their 'worlds' simplified to allow a novice or intermediate programmer to deal with the programming issues at hand without becoming bogged down in issues that essentially deal with matters of degree, rather than kind.

Alternately, some of the "simpler" games can still be appropriate for more advanced classes by focusing on animation, or graphics, or multi-player modes. The problem of saving state, even for a text-based action game can become a problem in file formats or data architecture.

Virtually all games listed here can be staged. Problems that can be staged are those that allow for varying levels of completion within the same assignment. For example, a game like *Frogger* allows for multiple stages of completion with even the simplest level having the attributes of a working game. *Frogger* is a particularly good example for it is currently playing a dual role as both classic arcade game, and lightweight console game. In *Frogger*, a low-level but still working solution would have only a single Frog that moves correctly on the screen, 2 rows of vehicles moving in opposite directions along the highway and one home at the top. There is no boulevard or river in this solution. The midrange solution will have 3 homes, 5 rows of vehicles, and a working Frog who can move (but not necessarily jump) and ONE OF: two kinds of river beast, ---OR--- a boulevard to rest on (with NO time limit for the Frog's stay). The Frog should be able to ride on the critters in the river instead of sliding off. The best solution will have 5 homes, 5 rows of vehicles, one Frog, AND 5 rows of river beasts all working correctly. The boulevard will have a time limit, and the turtles must sometimes dive. The "full-function" *Frogger* is bonus, and can be offered as a challenge for more advanced students. It includes the girl Frog; alligators whose mouths open, and snakes on logs and on the boulevard. Frogger himself should be permitted multiple incarnations.

Note that each stage introduces not only a new level of complexity to the gameplay, but, more importantly from the perspective of its value as a programming assignment, introduces additional complexity to the programming in the form of new algorithms.

8. Conclusions

Arcade games have a great deal to offer as subjects for programming assignments. They encompass all of the elements necessary for a fundamental grounding in computer science as well as many aspects of more advanced study, regardless of the student's eventual application area. Classic arcade games are especially suited to this task. Being part of the popular culture, these games are readily recognizable cultural objects, giving them a built-in connection to the real world, thus creating the authenticity necessary for effective student engagement.

A key requirement in the solution of any problem is to fully understand that problem. Having a working example of a solution with which students can interact is important. Having multiple examples is even better, but creating these for a newly made-up problem is time consuming. Multiple working examples of classic arcade games already exist in the public domain. Modern games are typically very complex and take full advantage of the latest developments in hardware resources. Having been created with twenty-five year old technology, classic games are

perforce less sophisticated then newer games. This means that recreating these objects is within the reach of novice programmers, while modern games are generally not (*Halo 2*, anyone?).

To close with one final thought, the more games get used as pedagogical tools, the more they will gain in general acceptance. This is almost certain to be a good thing.

9. Appendix: Classic Games and What We Can Learn with Them

Classic Arcade Games

1 = 1st year; **2** = 2nd year; **3+** = Senior level or Grad Course **MP** = can be designed as multiplayer

A/T = ASCII or Text-based; **A/G** = Animation and Graphics; **T/P** = Trajectories and other Physics; **IM** = Inventory Management

Y = Yes (blank = no) **N** = Novice, **I** = Intermediate, **A** = Advanced

Game	Type of game	Suitable for:			Pedagogy					
		1	2	3+	A/T	A/G	T/P	IM	MP	Other algorithms
Asteroids!	Action shooter	Y	Y	Y	Y	N,I,A	Y		Y	Collision, AI
Attaxx	Reversi		Y	Y	Y	N		Y	Y	AI
Blocks, Break Out	Bouncing object		Y	Y	Y	N	Y	Y	?	
Qbert	Puzzle	Y	Y		-					
Frogger	Simple strategy	Y	Y		Y					
Indy 500, Street racer	Racing		Y	Y	Y	IA	Y		Y	Collision, path-finding, AI
Lunar Lander	Gravity, physics		Y	Y	Y	NIA	Y			
Minesweeper, Gold Monkey	Grid puzzle	Y	Y		Y			Y	Y	Flood-fill
Missile Command, Defender	Action shooter, vector		Y	Y	Y	NIA	Y	Y	Y	Distance, AI
Pac Man	Maze		Y	Y	Y	N		Y		Path-finding, chasing
Donkey Kong, Joust, Mario Bros., Pitfall	Platform action adventure, side-scrolling		Y	Y	Y	NIA	?	Y	Y	Chasing, AI
Pong	Bouncing object	Y	Y	Y	Y	N	Y		Y	
Space Invaders, Space war	Shooter Vector		Y	Y	Y	NI	Y	Y	?	
Tetris	Puzzle, gravity, packing	Y	Y	Y	Y	N,I				Packing
Zork	Text-based adventure		Y	Y	Y			Y	Y	Parsing, AI

10. References

[1] J. Vegso, "Interest in CS as a Major Drops Among Incoming Freshmen," in Computing Research News, vol. 17: Computing Research Association, 2005, URL: http://www.cra.org/CRN/articles/may05/vegso.

[2] S. Hamm, "Home Is Where The Work Is," in Business Week Online, July 25, 2005 ed, 2005, URL: http://www.businessweek.com/magazine/content/05_30/b3944048_mz011.htm.

[3] "Computer Game Production Curriculum 2004 RFP Awards," in External Research and Programs: Mocrosoft Reserach, 2005, [Electronic Source] Retrieved from: http://research.microsoft.com/ur/us/fundingopps/Gaming_curriculumRFP_awards.aspx, accessed: July 28 2005.

[4] S. Cooper, W. Dann, and R. Pausch, "Teaching Objects-first In Introductory Computer Science," presented at 34th SIGCSE Technical Symposium on Computer Science Education, Reno, Navada, USA, 2003, URL: http://portal.acm.org/citation.cfm?id=611966.

[5] W. E. Doll, A post-modern perspective on curriculum. New York: Teachers College Press, 1993.

[6] J. Lave and E. Wenger, Situated learning: legitimate peripheral participation. Cambridge [England]; New York: Cambridge University Press, 1991.

[7] J. S. Bruner, Toward a theory of instruction. sl: sn., 1966.
[8] E. Roberts, "The dream of a common language: The search for simplicity and stability in computer science education.," presented at Thirty-Fifth SIGCSE Technical Symposium on Computer Science Education, Norfolk, VA, 2004, URL: http://doi.acm.org/10.1145/971300.971343.
[9] "Computing Curricula 2001: Final Report of the Joint ACM/IEEE-CS Task Force on Computer Science Education," IEEE Computer Press, Los Alamitos, CA December 2001, URL: http://www.acm.org/sigsce/cc2001.
[10] E. Soloway, "Learning to Program = Learning to Construct Mechanisms and Explanations," Communications of the ACM, Vol. 29 No. 9, pp. p850-858, 1986.
[11] K. Becker, "Teaching with games: the Minesweeper and Asteroids experience," Journal of Computing in Small Colleges, Vol. 17 No. 2, pp. 23-33, 2001, URL: http://www.cpsc.ucalgary.ca/~becker/Main/Papers/Asteroids.htm.
[12] E. R. Berlekamp, J. H. Conway, and R. K. Guy, Winning ways, for your mathematical plays. London: Academic Press, 1982.
[13] J. R. Parker and K. Becker, "The Use of Games in the Undergraduate Computer Science Curriculum." Calgary, 2002, [Electronic Source] Retrieved from: http://www.ucalgary.ca/~jparker, accessed: July 20 2005.
[14] J. R. Parker, Start Your Engines: Developing Racing and Driving Games. Scotsdale, AZ: Paraglyph Press, 2005.

First Year Programming Assignments:
(note: if anyone has difficulty accessing these, please send email to the author.)
Minesweeper Assignment:
http://pages.cpsc.ucalgary.ca/~becker/235/Asst/MineSweeper/MineSweeper.html
Asteroids Assignment:
http://pages.cpsc.ucalgary.ca/~becker/235/Asst/Asteroids/Asteroids.html
Space Invaders Assignment:
http://pages.cpsc.ucalgary.ca/~becker/235/Asst/SpaceInvaders/Invaders.html
Centipede Assignment:
http://pages.cpsc.ucalgary.ca/~becker/235/Asst/Centipede/Centipede.html
Tetris Assignment:
http://pages.cpsc.ucalgary.ca/~becker/235/Asst/Tetris/tetris.htm
Frogger Assignment:
http://pages.cpsc.ucalgary.ca/~becker/235/Asst/Frogger/Frogger.html